МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ

КАФЕДРА СИСТЕМНОГО ПРОЕКТУВАННЯ

**КУРСОВА РОБОТА**

з дисципліни «Алгоритмізація та програмування»

на тему: Програма роботи з матрицями

Виконав: студент 1 курсу групи ДА-12

спеціальність 122 «Комп’ютерні науки»

Ніколаєв Роман Юрійович  
Керівник: Романов В.В ⠀  
Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_Оцінка: ECTS \_\_\_\_

Прийняли:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
(підпис) (, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) ( прізвище та ініціали)

Засвідчую, що у цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань

студент \_\_\_\_\_\_\_\_\_\_\_\_

Київ - 2022 рік

Національний технічний університет України “КПІ” ННК “ІПСА”

(назва вищого навчального закладу)

Кафедра системного проектування ⠀

Дисципліна алгоритмізація та програмування ⠀

Спеціальність 122 Комп’ютерні науки ⠀

Курс I ⠀Група ДА-12 Семестр 2 ⠀

**ЗАВДАННЯ**

**на курсову роботу студента**

Ніколаєв Романа Юрійовича

(прізвище, ім’я, по батькові)

1. Тема роботи

Швидке сортування даних

2. Строк здачі студентом закінченого проекту (роботи) .05.22 ⠀

3. Вихідні дані до проекту (роботи)

Мова програмування С++

Програма для ОС Linux

Технічна література

4. Зміст розрахунково – пояснювальної записки (перелік питань, які підлягають розробці)

Постановка задачі

Розробка програмного продукту

Опис розробленого програмного продукту

Висновки

Список літератури

5. Перелік графічного матеріалу ( з точним зазначенням обов’язкових креслень)

Блок-схеми алгоритмів

Знімки екрану додатка

6. Дата видачі завдання 08.02.2022 ⠀

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Найменування етапів курсової роботи | Строк виконання  етапів роботи | Примітки |
| 1. | Вивчення та аналіз алгоритму | 18.03.2022 |  |
|  |  |  |  |
| 2. | Визначення структури програм- них модулів | 06.04.2022 |  |
|  |  |  |  |
| 3. | Розробка інтерфейсу | 15.04.2022 |  |
|  |  |  |  |
| 4. | Оптимізація програми | 29.04.2022 |  |
|  |  |  |  |
| 5. | Налагодження та тестування  програми | 04.05.2022 |  |
|  |  |  |  |
| 6. | Оформлення роботи | 10.06.2022 |  |
|  |  |  |  |
| 7. | Здача на перевірку (електронна версія | 17.06.2022 |  |
|  |  |  |  |
| 8. | 8. Виправлення недоліків та доробка програми |  |  |
|  |  |  |  |
| 9. | Захист |  |  |

Студент ⠀ Ніколаєв Р. Ю. ⠀   
 (підпис студента) (прізвище, ім‘я, по батькові студента)  
Керівник ⠀ Романов В.В ⠀   
 (підпис викладача) (прізвище, ім‘я, по батькові викладача)

Зміст

ВСТУП..................................................................................................5  
**1.**ОБҐРУНТУВАННЯ ТА ВИБІР АЛГОРИТМІВ....................................6  
 **1.1.**Quick Sort................................................................................6 **1.2.**Merge Sort...............................................................................8 **1.3.**Heap Sort.................................................................................9  
**2.**РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ.........................................11  
 **2.1.**Загальні відомості..................................................................11  
 **2.2.**Функціональне призначення..................................................11  
 **2.3.**Опис логічної структури........................................................12  
 **2.4.**Вхідні дані.............................................................................14  
 **2.5.**Вихідні дані...........................................................................15  
**3.**ОПИС ПРОГРАМНОГО ПРОДУКТУ................................................16  
 **3.1.**Керівництво користувача.......................................................16  
 **3.2.**Керівництво розробника.........................................................18  
 **3.3.**Результати..............................................................................18

4.СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ...............................................21  
ВИСНОВКИ..........................................................................................22  
ДОДАТКИ.............................................................................................22  
Додаток А. Лістинг програми...............................................................22  
Додаток Б. Блок-схеми.........................................................................35

**Вступ**

Потреба у ефективному сортуванні великих об’ємів даних пояснюєтьсяя поширенням ЕОМ в середині ХХ ст., для вирішення котрої була розроблена низка алгоритмів, такі як Quick Sort, Merge Sort, Heap Sort, котрі широко використовуються і сьогодні. Неможливо уявити популярний мобільний додаток, чи складну виробничу систему без алгоритмів сортування.

Власне сортування *(англ. Sort – впорядковувати)* даних це процес, під час якого, заданий об’єм даних впорядковується певним критерієм, завдяки виконаним операціям над данимим (порівняння, зміна місцями). Наприклад, в мобільному додатку це сортування аккаунтів по рейтингу, чи сортування задач на виробництві за приорітетністю, тощо. Отже, при винекненні ефективно впорядкувати великий об’єм даних, використовують алгоритми сортування, яких існує велика множина, кожен з яких має власні переваги і недоліки. Також алгоритми можна розділити за їх асимптотичною складністю. Асимптотична складність – це залежність часу виконання або використаної пам’яті комп’ютера при виконанні програми від кількості вхідних даних. Більшість алгоритмів сортування мають складність О(n2), де n – кількість вхідних даних, що є гарним варіантом для сортування невеликих об’ємів, але при великих значеннях n, ефективність сортування сильно падає, і є доречним розглянути алгоритми швидкого сортування.

Отже, оптимізація використання наявних ресурсів при сортуванні великих об’ємів даних є ключовим фактором при розробці відповідних алгоритмів, що і змотивувало мене дослідити найпопулярніші алгоритми швидкого сортування даних та дослідити їх на ефективність. .

# **1. Обґрунтування та вибір алгоритмів**

Для представлення даних був обраний, швидкий та простий у використанні метод – динамічний масив. Масивом є послідовна і неперервна послідовність даних, розташована у комірках оперативної пам’яті, з миттєвим доступом за індексом відповідного елементу, що робить його ідеальним варіантом для демонстрації алгоритмів швидкого сортування, оскільки динамічний масив майже не впливає на виконання алгоритму.

При розробці програми використаємо готову реалізацію структури даних динамічний масив з бібліотеки STL. Масив можна заповнити вручну, даними з файлу або випадковими числами.

## **1.1. Quick Sort**

Алгоритм:

* Спочатку вибирається «опорний» елемент масиву, який може бути будь-яким елементом, але від вибору цього елементу залежить подальша ефективність виконання. Ідеальним варіантом «опори» є елемент, найближчий до медіани.
* Масив умовно розбивається на дві частини відносно «опори» – ліворуч елементи менше «опорного», праворуч – більше. Таким чином ми отримуємо дві половини одного масиву, для кожної з котрих операціїї по розділу рекурсивно повторюються.

Псевдокод:

QuickSort(array, pos1, pos2):  
 if pos1 >= pos2 return

i = divide(array, pos1, pos2)

QuickSort(array, pos1, i-1)

QuickSort(array, i+1, pos2)

array – масив, pos1 – початковий індекс, pos2 – кінцевий індекс, i – індекс опорного елемента;

Допоміжна функція

divide(array, pos1, pos2):

x = array[pos1]

i = pos1-1

for j = pos1 to pos2 -1

Do if array[j] <= x

then

i = i+1  
 swap (array[i], array[j])

swap(array[i+1], array[pos2])

return i+1

array - масив, pos1 – початковий індекс, pos2 – кінцевий індекс, i - індекс опорного елемента;

Розгляд можливих випадків асимптотичної складності:

* Найгірший

Якщо масив відсортований по спаданню, то при рекурсивному повторенню, масив розбивається таким чином, що одна частина не містить елементів, а інша містить всі елементи, окрім опірного, і тільки на це необхідно О(n) операцій, що виливається в загальну складність О(n2).

* Найкращий та середній

Середній та найкращий випадки є наближеними та відбуваються за умови якщо масив при кожному розділенні масиву, від ділить на два (розмір кожної частини не перевищує n/2), то асимптотичною складністю є О(n\*log(n)).

## **1.2. Merge Sort**

Алгоритм:

* Рекурсивно розбиваємо масив на дрібні частини однакового розміру, доки не зникнуть всі окремі елементи
* Зливаємо сусідні масиви в один: вибираємо найменший вказівник на початок масиву з двох сусідніх; у масиві, в якому було менше число покажчик переноситься на наступний елемент, повторюємо попередній пункт; якщо в одному з масивів закінчились елементи – переносимо елементи іншого масиву до відсортованого масиву
* Повторюємо, доки всі масиви не зіллються в один

Псевдокод:

MergeSort(array, pos1, pos2)

if pos2 - pos1 < 1

then return

base = (pos1-pos2)/2

if pos2 - pos1 > 1

MergeSort(array, pos1, base)

MergeSort(array,base+1, pos2)

Merge(array, pos1, pos2, base)

Де array - масив, pos1 – початковий індекс, pos2 – кіецевий індекс, base – індекс центрального елемента.

Merge(array, pos1, pos2, base)

i = pos1

j = base+1

for k=pos1 to pos2

do if j > pos2 or (i <= base and array[i] <= array[j])

then tempArray[k] = A[i]

i++

else tempArray[k] = array[j]

J++

for k = pos1 to pos2

do array[k] = tempArray[k]

Де array - масив, tempArray – тимчасовий масив, pos1 – початковий індекс, pos2 – кіецевий індекс, base – індекс центрального елемента.

Асимптотична складність – О(n\*log(n)). Алгоритм не міняє порядок розташування однакових елементів, а отже він є стабільним.

**1.3. Heap Sort**

Алгоритм:

* HeapSort використовує бінарне дерево сортування, що представляє собою абстрактну структуру даних, яка вважається правильною за деяких умов:
* Кожен лист має глибину d (d-1), де d – максимальна глибина дерева
* Значення в будь-якій вершині не менші(більше) за значення їх нащадків
* Спочатку елементи масиву вибудовуються у вигляді бінарного дерева, що потребує N операцій.
* Після будування дерева, елементи з кореня послідовно видаляються і все дерево «просіюється» щоб задовольняти умови «правильності»

Псевдокод:

HeapSort(array):

Heap(array) //будування дерева

for i = len(arra) to 2

do swap array[i],array[i]

size- -

Heapify(array, i)

Heap(array):

size = length(array)

for i = size/2 to 1

do Heapify(array, i)

Heapify(array, i):

pos1 = 2 \* i + 1

pos2 = 2 \* i + 2

if(pos1 <= size) and (array[pos1]>array[i])

largest = pos1

else

largest = i

if(pos2 <= size) and (array[pos2] > array[largest])

swap(array[i], array[largest])

Heapify(arry, largest)

Де array - масив.

Асимптотична складність – О(n\*log(n)) в середньому.

**2. Розробка програмного продукту**

## **2.1. Загальні відомості**

Створена програма була названа “SortingAlgorithms” і представляє з себе консольний додаток. Написана була на мові С++ стандарту С++ 14, за допомогою IDE JetBrains Clion. Для комфортного використання, комп’ютер користувача повинен задовольняти певним мінімальним вимогам:  
 ОС – Linux

Процесор - 1 ГГц

Графічна карта - 100 Мб відеопам’яті

Оперативна пам’ять - 512 Мб

Диск - 10 Мб вільної пам’яті

**2.2. Функціональне призначення**

Основне призначення – демонстрація можливостей алгоритмів швидкого сортування. Користувач має змогу ввести дані для сортування у зручний йому спосіб (або згенерувати їх випадковим чином) та відсортувати введені дані одним з трьох представлених алгоритмів сортування.

Відсортовані дані можна зберегти у файл, шлях, назву та розширення якого, користувач має змогу обрати самостійно.

Також програма здатна згенерувати демонстративний файл зі статистикою роботи алгоритмів, що зручно для побудови графіків та перевірки асимптотичної складності виконання в залежності від кількості вхідних даних.

**2.3. Опис логічної структури**

Даний розділ описує головні функції програми.

void fillFromFile(std::vector<int>& array, std::string fileName);

Функція приймає векторний масив та ім’я файлу, з котрого дані будуть взяті для запису у масив для подальшого опрацювання.

Див. блок-схему 1.

void saveArray(std::vector<int>& array);

Функція приймає векторний масив, та запрошує у користувача назву файлу, котрий буде створений, і в котрий буде записаний вміст масиву.

Див. блок-схему 2.

\*примітка: деякі алгоритми швидкого сортування рекурсивні, тож для зручності використання іншим розробником були створені “зовнішні” функції, котрі приймають мінімум аргументів, а вже потім викликають основну рекурсивну функцію сортування з необхідними алгоритмами.

Приклад:

void quickSort(std::vector<int>& array); викликаєvoid quickSortCall(std::vector<int> &array, int pos1, int pos2)

void quickSortCall(std::vector<int> &array, int pos1, int pos2)

Приймає на вхід векторний масив для сортування та два числа типу int, які є індексами елементів, між якими відбудеться сортування.

Див. блок-схему 3.

void mergeSortCall(std::vector<int> &array, int pos1, int pos2)

Приймає на вхід векторний масив для сортування та два числа типу int, які є індексами елементів, між якими відбудеться сортування.

Див. блок-схему 4.

void merge(std::vector<int> &array, int pos1, int pos2)

Використовується у mergeSortCall() та ділить масив на дві частини. Приймає на вхід векторний масив для сортування та два числа типу int, які є індексами елементів, між якими відбудеться сортування.

Див. блок-схему 5.

void sa::heapSort(std::vector<int> &array)

Приймає на вхід векторний масив для сортування.

Див. блок-схему 6.

void heapify(std::vector<int> &array, int i, int size)

Використовується у heapSort() та приймає на вхід векторний масив, ціле число і та ціле число size, що є розміром масиву.

Див. блок-схему 7.

\*примітка: далі буде розглянуто три однотипні функції, котрі мають ідентичну структуру, але різну назву та виклик відповідного алгоритму сортування всередині. Мова про getTimeQuickSort(), getTimeMergeSort(), getTimeHeapSort(), далі - getTime[Сортування]().

float getTime[Сортування](int amount, int times)

Функція повертає значення типу float, що є загальним часом впорядкування amount елементів times разів за допомогою відповідного алгоритму сортування. Приймає на вхід цілі числа amount та times. Використовується у getAlgorithmsData().

Див. блок-схему 8.  
  
void test::getAlgorithmsData()  
 Функція для отримання статистики про роботу всіх трьох алгоритмів швидкого сортування та її запису у файл.

Файл заповнюється у форматі [amount,time; ], де amount – кількість елементів, що підлягли сортуванню, time – час, за який масив був відсортований (див. Рисунок 8).

Див. блок-схему 9.

**2.4. Вхідні дані**

Вхідними даними є масив цілих чисел, котрий користувач задає самостійно запропонованими програмою методами. Це може бути: автоматичне випадкове заповнення масиву зазначеного розміру, ручне заповнення масиву зазначеного розміру, або заповнення з існуючого .txt файлу.

Якщо при введенні даних шляхом файлу, формат заповнення або тип даних не є правильними, то поведінка програми є невизначеною.

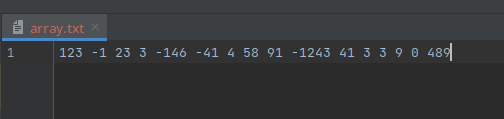


Рисунок 1 – приклад правильно заповненого файлу для введення даних

Якщо при введенні даних шляхом ручного вводу, тип даних не є правильним, то поведінка програми також є невизначеною.

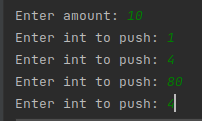


Рисунок 2 – приклад правильного заповнення масиву через консоль

**2.5. Вихідні дані**

Вихідними даними є раніше введений масив у впорядкованому вигляді. Отримати дані можна безпосередньо в консолі, чи створити окремий .txt файл, куди буде записаний результуючий масив.

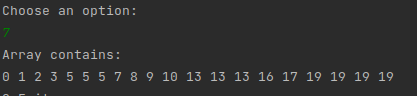


Рисунок 3 – приклад виведення відсортованого масиву у консоль

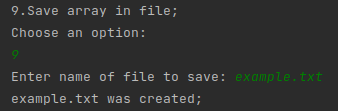


Рисунок 4 – збереження масиву у файл «example.txt»

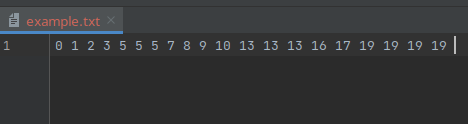


Рисунок 5 – файл «example.txt»

**3. Опис програмного продукту**

## **3.1. Керівництво користувача**

Для початку користування програми на ОС Linux необхідно запустити файл “SortingAlgorithms” в консолі. Це можна зробити, натиснувши по файлу ПКМ та вибравши пункт “Run In Konsole” у випадаючому контекстному меню.

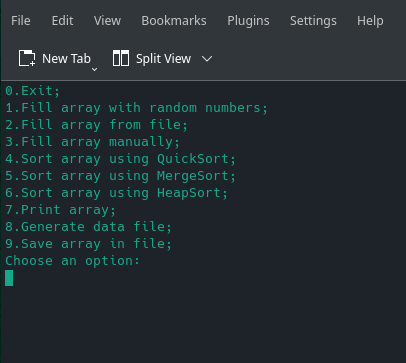


Рисунок 6 – вигляд програми “SortingAlgorithms” у консолі

Програма має меню з десяти пунктів:

***0.Exit*** – завершення роботи програми.

***1.Fill array with random numbers*** – програма запросить у користувача бажаний розмір майбутнього масиву і створить його, заповнивши випадковими значеннями.

***2.Fill array from file*** – програма запропонує користувачу ввести назву файлу, з котрого необхідно взяти дані. Якщо файлу не існує – програма сповістить користувача про це.

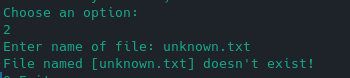


Рисунок 7 – приклад помилки знаходження файлу

Важливо зазначити, що якщо формат заповнення файлу не буде дотриманий, то поведінка програми не є визначеною (див. Рисунок 1).

***3.Fill array manually*** – програма запросить у користувача бажаний розмір майбутнього масиву і запустить форму для послідовного заповнення.

***4.Sort array using QuickSort*** – сортування масиву алгоритмом QuickSort.

***5.Sort array using MergeSort*** - сортування масиву алгоритмом MergeSort.

***6.Sort array using HeapSort*** - сортування масиву алгоритмом HeapSort.

***7.Print array*** – вивід змісту масиву у консоль (див. Рисунок 3).

***8.Generate data file*** – створення файлу “data.txt” з актуальною статистичною інформацією про роботу алгоритмів.

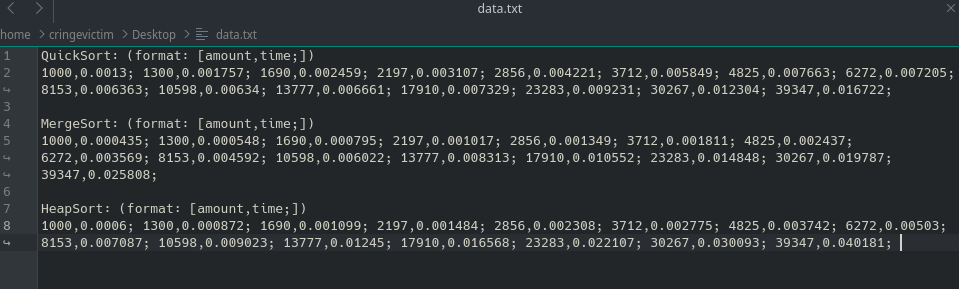


Рисунок 8 – файл “data.txt”

***9.Save array in file*** - програма запропонує користувачу ввести назву файлу, котрий буде створений із записаним у нього поточним масивом.

В майбутньому, цей самий файл можна використати в якості даних для введення (див. Рисунок 4).

**3.2. Керівництво розробника**

Дана реалізація алгоритмів сортування є достатньо розповсюдженою і надійною. Асимптотична складність кожного з них складає О(n\*log(n)).

Основний код винесен в окремі файли, тож, за потреби, легко піддається модернізації. Наприклад, можна використати методи шаблонних функцій та перенавантаження операторів порівняння, щоб дані алгоритми мали змогу опрацьовувати абсолютно різні типи даних.

**3.3. Результати**

Одна з цілей даної роботи – дослідження асимптотичної складності алгоритмів швидкого сортування, для чого і була розроблена функція getAlgorithmsData(), результат роботи якої був використаний для побудови графіків продуктивності алгоритмів.

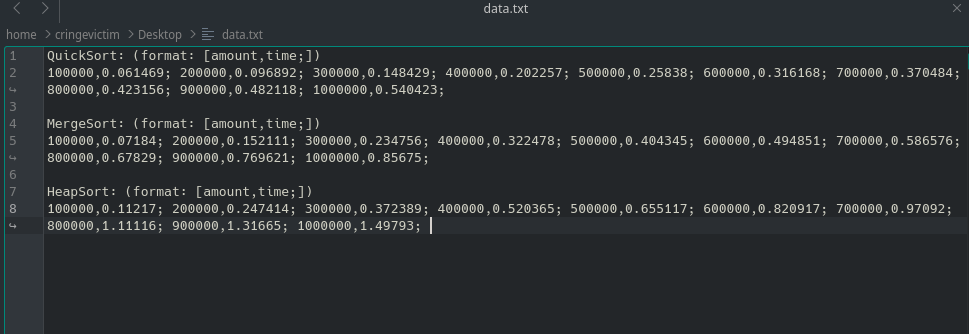


Рисунок 9 – файл “data.txt”, як результат виконання getAlgorithmsData()

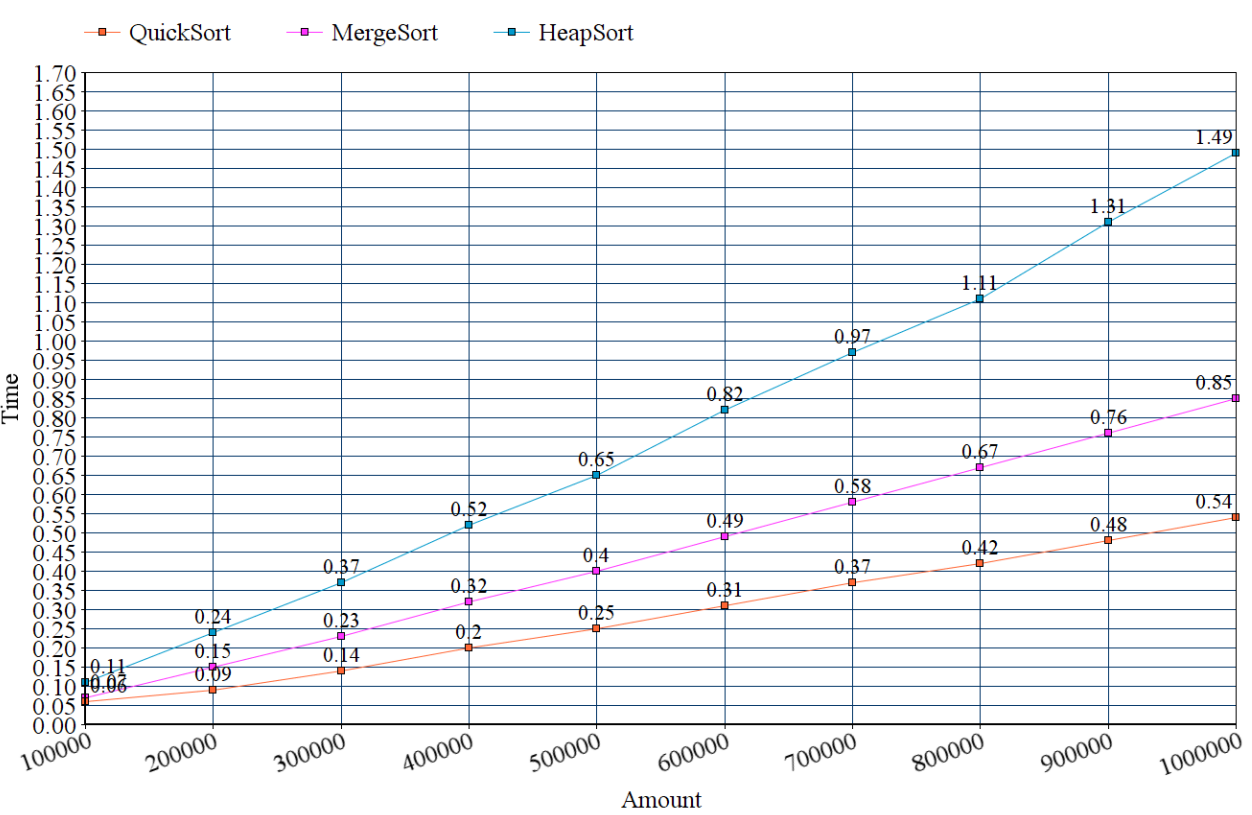


Рисунок 10 – графіки продуктивності алгоритмів швидкого сортування

Як ми можемо спостерігати, графік QuickSort виявляється найнижче, тобто найефективніше серед всіх, якщо розглядати середній або найкращі випадки.

Але не слід забувати про найгірший випадок, за якого асимптотика даного алгоритму зростає до O(n2) і даний алгоритм стає найповільнішим серед переліченої трійки. Це відбувається за попередньої відсортовки масиву у зворотньому порядку.

На рисунку 11 представлений графік залежності *y = x\*log(x),* що є заявленою асимптотичною складністю кожного з трьох алгоритмів швидкого сортування у середніх та найкращих випадках.

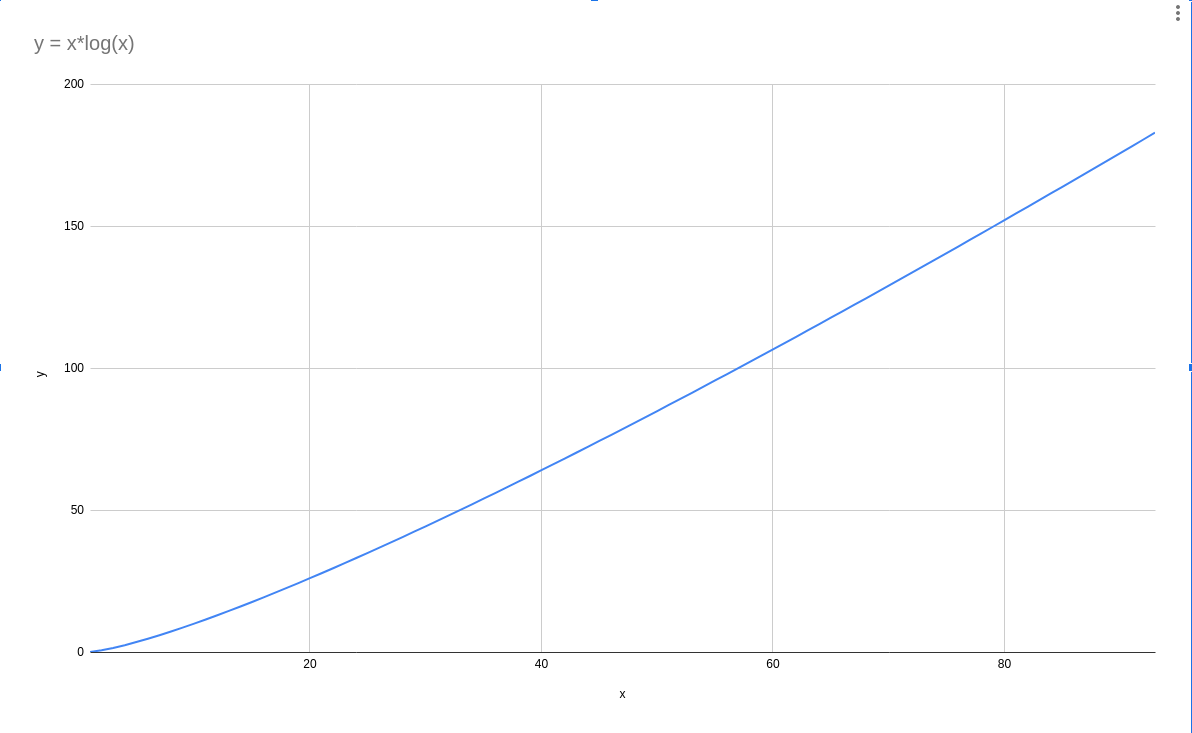


Рисунок 11 - *y = x\*log(x)*

Неможливо не помітити, що графіки продуктивності і графік математичної залежності візуально наближені один до одного. З цього можна зробити висновок, що заявлена асимптотична складність відповідає дійсності. Похибку можна списати на особливості кожного конкретного алгоритму та нерівномірність в швидкості обрахунків.

# **Висновки**

У ході виконання даної роботи, за допомогою мови програмування С++ стандарту С++ 14 та IDE JetBrains Clion, було розроблено програму для опрацювання вхідних даних за допомогою алгоритмів швидкого сортування QuickSort, MergeSort, HeapSort. Також було розроблено методи для дослідження їх ефективності.

Розглянемо кожен алгоритм окремо:

**HeapSort:**

З графіку видно, що це найповільніший з представлених алгоритмів сортування, але, на відміну від MergeSort та QuickSort, він не вимагає масивного рекрсивного виклику чи декількох масивів для роботи.

**MergeSort:**

Сортування злиттям є трохи швидшим, ніж HeapSort, але для цього необхідно вдвічі більше виділеної пам’яті, якщо судити по структурі самих алгоритмів.

**QuickSort:**

Швидке сортування можна назвати швидшою версією MergeSort, проте ефективність алгоритму сильно залежить від порядку розташування елементів в масиві, а також від того, який самий елемент обрано в якості опорної точки.

Підбиваючи підсумки, можна сказати, що універсального алгоритму не існує і розробник повинен чітко розуміти з чим він працює і підбирати алгоритм відповідно до поставленої задачі. Розглядаючи дану трійку доводиться шукати компроміс між ефективністю по пам’яті (HeapSort), стабільністю (MergeSort) та швидкістю в конкретних випадках (QuickSort).

**Список використаних джерел**

1. Інтернет-енциклопедія “Вікіпедія”. Стаття “Швидке сортування даних”. URL: <https://uk.wikipedia.org/wiki/%D0%A8%D0%B2%D0%B8%D0%B4%D0%BA%D0%B5_%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F>
2. Інтернет-енциклопедія “Вікіпедія”. Стаття “Сортування злиттям”.

URL: <https://uk.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B7%D0%BB%D0%B8%D1%82%D1%82%D1%8F%D0%BC>

1. Інтернет-енциклопедія “Вікіпедія”. Стаття “Пірамідальне сортування”. URL: <https://uk.wikipedia.org/wiki/%D0%9F%D1%96%D1%80%D0%B0%D0%BC%D1%96%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B5_%D1%81%D0%BE%D1%80%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F>

## **Додаток А. Лістинг програми**

**Файл “main.cpp”:**

//Program:"SortingAlgorithms"

//Developer:Nikolaiev Roman Yurievich

//Date: 10/06/22

//Purpose:Data sorting and complexity proving

//Limits:Not recommended to use arrays longer than 10^8 elements

#include <iostream>

#include <vector>

#include <ctime>

#include "SortingAlgorithms.h"

#include "SortData.h"

#include "ArrayMethods.h"

int main()

{

srand(time(nullptr));

int limit = 10^7;

std::vector<int> array;

std::string fileName;

clock\_t startTime, endTime;

float fullTime;

int amount = 20;

int choice = 99;

while(choice){

std::cout << "0.Exit;"

"\n1.Fill array with random numbers;"

"\n2.Fill array from file;"

"\n3.Fill array manually;"

"\n4.Sort array using QuickSort;"

"\n5.Sort array using MergeSort;"

"\n6.Sort array using HeapSort;"

"\n7.Print array;"

"\n8.Generate data file;"

"\n9.Save array in file;" << std::endl;

std::cout << "Choose an option: " << std::endl;

std::cin >> choice;

switch(choice){

case 1:

std::cout << "Enter amount: ";

std::cin >> amount;

am::fillVecRandomInt(array, amount);

std::cout << "Array filled successfully;" << std::endl;

break;

case 2:

std::cout << "Enter name of file: ";

std::cin >> fileName;

am::fillFromFile(array, fileName);

break;

case 3:

std::cout << "Enter amount: ";

std::cin >> amount;

am::fillManually(array, amount);

std::cout << "Array filled successfully;" << std::endl;

break;

case 4:

startTime = clock();

sa::quickSort(array);

endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

std::cout << "Time: " << fullTime << std::endl;

std::cout << "Array sorted successfully;" << std::endl;

break;

case 5:

startTime = clock();

sa::mergeSort(array);

endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

std::cout << "Time: " << fullTime << std::endl;

std::cout << "Array sorted successfully;" << std::endl;

break;

case 6:

startTime = clock();

sa::heapSort(array);

endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

std::cout << "Time: " << fullTime << std::endl;

std::cout << "Array sorted successfully;" << std::endl;

break;

case 7:

am::printIntVec(array);

break;

case 8:

test::getAlgorithmsData();

break;

case 9:

am::saveArray(array);

break;

default:

break;

}

}

}

**Файл “SortingAlgorithms.h”:**

#pragma once

#include <vector>

namespace sa{

void quickSort(std::vector<int>& array);

void mergeSort(std::vector<int>& array);

void heapSort(std::vector<int>& array);

} **Файл “SortingAlgorithms.cpp”:**

#include <iostream>

#include "SortingAlgorithms.h"

void quickSortCall(std::vector<int> &array, int pos1, int pos2) {

{

int i = pos1;

int j = pos2;

int base = array.at((i + j) / 2);

int temp;

while (i <= j) {

while (array.at(i) < base)

i++;

while (array.at(j) > base)

j--;

if (i <= j) {

temp = array.at(i);

array.at(i) = array.at(j);

array.at(j) = temp;

i++;

j--;

}

}

if (j > pos1)

quickSortCall(array, pos1, j);

if (i < pos2)

quickSortCall(array, i, pos2);

}

}

void merge(std::vector<int> &array, int pos1, int pos2)

{

int i = pos1,

t = 0,

mid = pos1 + (pos2 - pos1) / 2,

j = mid + 1,

d[array.size()];

while(i <= mid && j <= pos2){

if(array.at(i) <= array.at(j)) {

d[t] = array.at(i); i++;

}

else{

d[t] = array.at(j); j++;

}

t++;

}

while(i <= mid){

d[t] = array.at(i); i++; t++;

}

while(j <= pos2){

d[t] = array.at(j); j++; t++;

}

for(i = 0; i < t; i++) array.at(pos1 + i) = d[i];

}

void mergeSortCall(std::vector<int> &array, int pos1, int pos2)

{

int temp;

if (pos1 < pos2)

if (pos2 - pos1 == 1) {

if (array.at(pos2) < array.at(pos1)){

temp = array.at(pos1);

array.at(pos1) = array.at(pos2);

array.at(pos2) = temp;

}

}

else {

mergeSortCall(array, pos1, pos1 + (pos2 - pos1) / 2);

mergeSortCall(array, pos1 + (pos2 - pos1) / 2 + 1, pos2);

merge(array, pos1, pos2);

}

}

void sa::quickSort(std::vector<int> &array) {

quickSortCall(array, 0, array.size()-1);

}

void sa::mergeSort(std::vector<int> &array) {

mergeSortCall(array, 0, array.size()-1);

}

void heapify(std::vector<int> &array, int i, int size)

{

int largest, l = (2\*i) + 1, r = l + 1;

if(l < size && array.at(l) > array.at(i))

largest = l;

else

largest = i;

if(r < size && array.at(r) > array.at(largest))

largest = r;

if(largest != i)

{

std::swap(array.at(i), array.at(largest));

heapify(array, largest, size);

}

}

void sa::heapSort(std::vector<int> &array) {

for(int i = (array.size() / 2); i >= 0; i--)

heapify(array, i, array.size());

int sz = array.size();

for(int i = array.size() - 1; i > 0; i--)

{

std::swap(array[0], array[i]);

sz--;

heapify(array, 0, sz);

}

} **Файл “SortData.h”:**

#pragma once

namespace test{

void getAlgorithmsData();

} **Файл “SortData.cpp”:**

#include <iostream>

#include <ctime>

#include <vector>

#include <fstream>

#include "ArrayMethods.h"

#include "SortData.h"

#include "SortingAlgorithms.h"

float getTimeQuickSort(int amount, int times){

std::vector<int> array;

float fullTime = 0;

for(int i = 0; i < times; i++){

am::fillVecRandomInt(array, amount);

clock\_t startTime = clock();

sa::quickSort(array);

clock\_t endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

}

return fullTime;

}

float getTimeMergeSort(int amount, int times){

std::vector<int> array;

float fullTime = 0;

for(int i = 0; i < times; i++){

am::fillVecRandomInt(array, amount);

clock\_t startTime = clock();

sa::mergeSort(array);

clock\_t endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

}

return fullTime;

}

float getTimeHeapSort(int amount, int times){

std::vector<int> array;

float fullTime = 0;

for(int i = 0; i < times; i++){

am::fillVecRandomInt(array, amount);

clock\_t startTime = clock();

sa::heapSort(array);

clock\_t endTime = clock();

fullTime += (float(endTime - startTime)) / CLOCKS\_PER\_SEC;

}

return fullTime;

}

void test::getAlgorithmsData() {

int reps = 10;

int X = 100000;

std::string fileName = "data.txt";

std::fstream file;

file.open(fileName, std::fstream::out);

if(file.is\_open()){

file << "QuickSort: (format: [amount,time;])\n";

for(int i = 0, amount = 100000; i < reps; i++, amount+=X){

file << amount << "," << getTimeQuickSort( amount, 1) << "; ";

}

file << "\n\nMergeSort: (format: [amount,time;])\n";

for(int i = 0, amount = 100000; i < reps; i++, amount+=X){

file << amount << "," << getTimeMergeSort( amount, 1) << "; ";

}

file << "\n\nHeapSort: (format: [amount,time;])\n";

for(int i = 0, amount = 100000; i < reps; i++, amount+=X){

file << amount << "," << getTimeHeapSort( amount, 1) << "; ";

}

} else {

std::cout << "Error occurred;" << std::endl;

file.close();

return;

}

std::cout << "Data file generated successfully;" << std::endl;

file.close();

} **Файл “ArrayMethods.h”:**

#pragma once

#include <vector>

#include <iostream>

namespace am{

void printIntVec(std::vector<int> array);

void fillVecRandomInt(std::vector<int>& array, int amount);

void fillManually(std::vector<int>& array, int amount);

void fillFromFile(std::vector<int>& array, std::string fileName);

void saveArray(std::vector<int>& array);

} **Файл “ArrayMethods.cpp”:**

#include <iostream>

#include <vector>

#include <fstream>

#include "ArrayMethods.h"

void am::printIntVec(std::vector<int> array) {

std::cout << "Array contains: " << std::endl;

for(int i = 0; i < array.size(); i++) std::cout << array.at(i) << " "; std::cout << std::endl;

}

void am::fillVecRandomInt(std::vector<int>& array, int amount){

if(!array.empty()) array.clear();

for(int i = 0; i < amount; i++) array.push\_back(rand()%amount);

}

void am::fillManually(std::vector<int> &array, int amount) {

if(!array.empty()) array.clear();

for(int i = 0, obj; i < amount; i++) {

std::cout << "Enter int to push: ";

std::cin >> obj;

array.push\_back(obj);

}

}

void am::fillFromFile(std::vector<int> &array, std::string fileName) {

if(!array.empty()) array.clear();

std::ifstream file(fileName);

int num;

if(file.is\_open()){

std::cout << "File contains: " << std::endl;

while (file >> num) {

array.push\_back(num);

std::cout << num << "; ";

}

std::cout << std::endl;

} else {

std::cout << "File named [" << fileName << "] doesn't exist!" << std::endl;

file.close();

return;

}

std::cout << "Array filled successfully;" << std::endl;

file.close();

}

void am::saveArray(std::vector<int> &array) {

std::string fileName;

std::cout << "Enter name of file to save: ";

std::cin >> fileName;

std::fstream file;

file.open(fileName, std::fstream::out);

if(file.is\_open()){

for(int i = 0; i < array.size(); i++){

file << array.at(i) << " ";

}

} else {

std::cout << "Error occurred;" << std::endl;

file.close();

return;

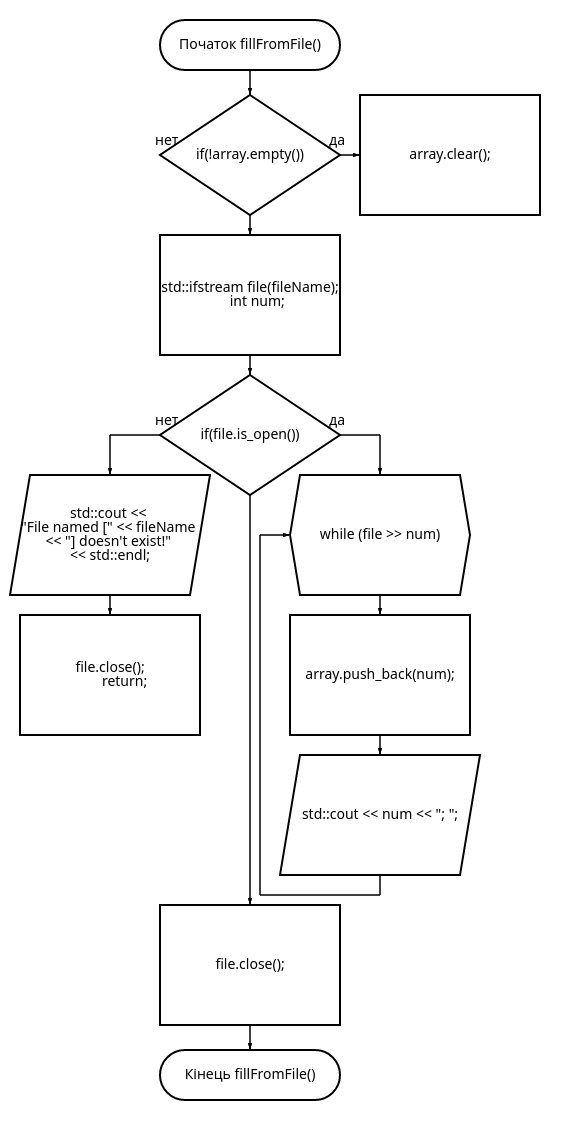
}

std::cout << fileName <<" was created;" << std::endl;

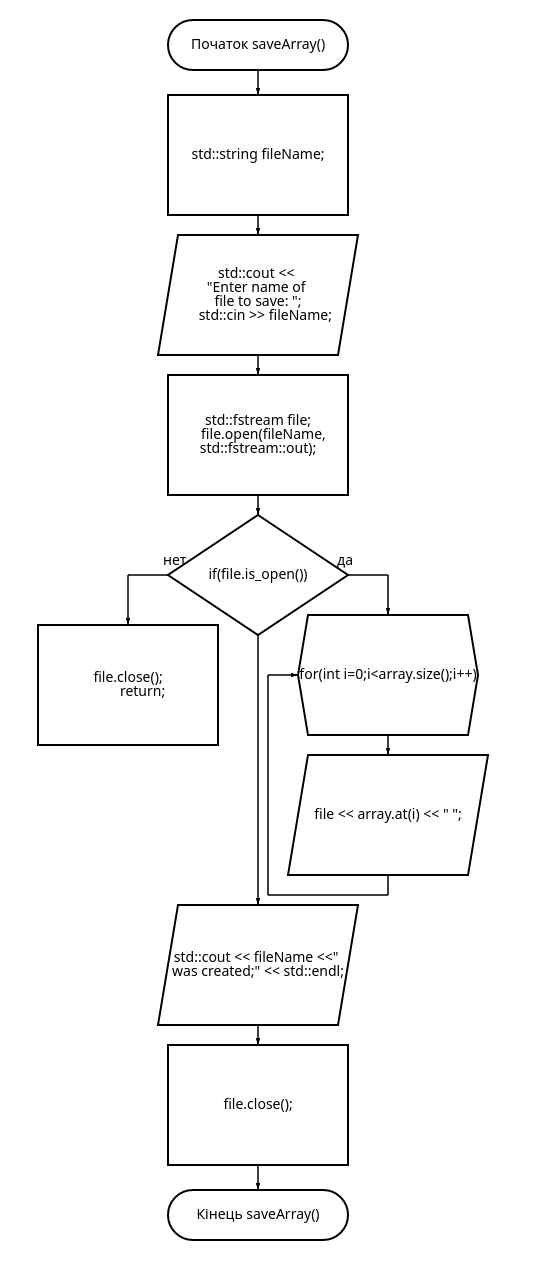
file.close();

}

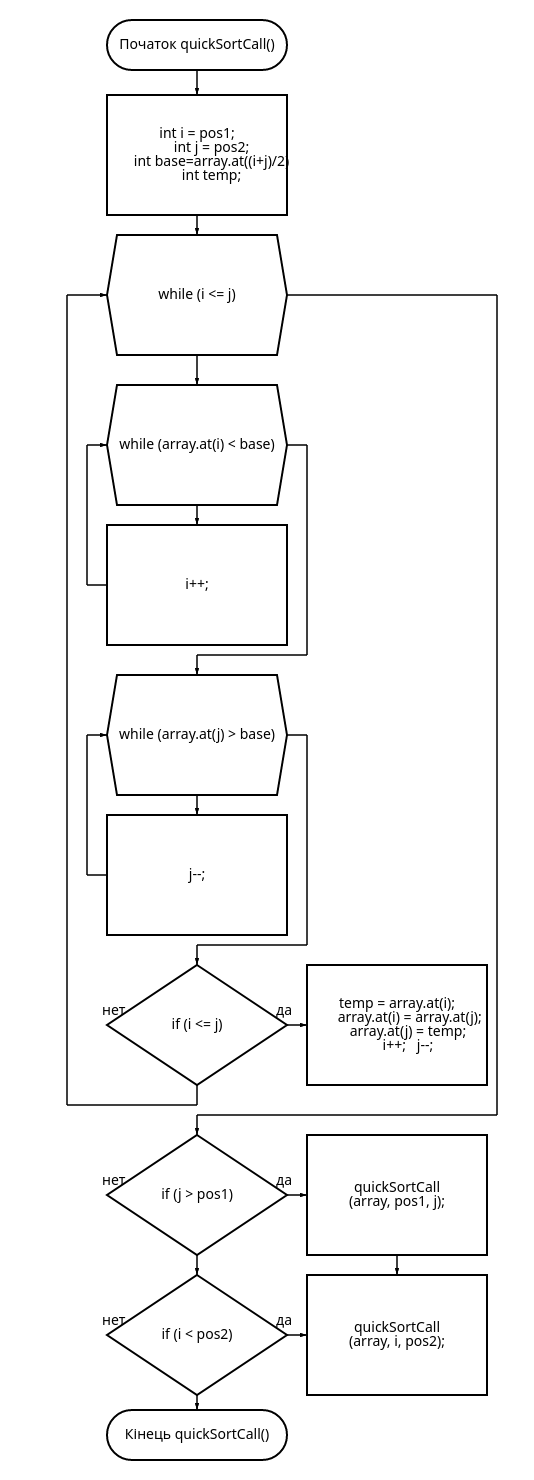
## **Додаток Б. Б****лок-схеми**



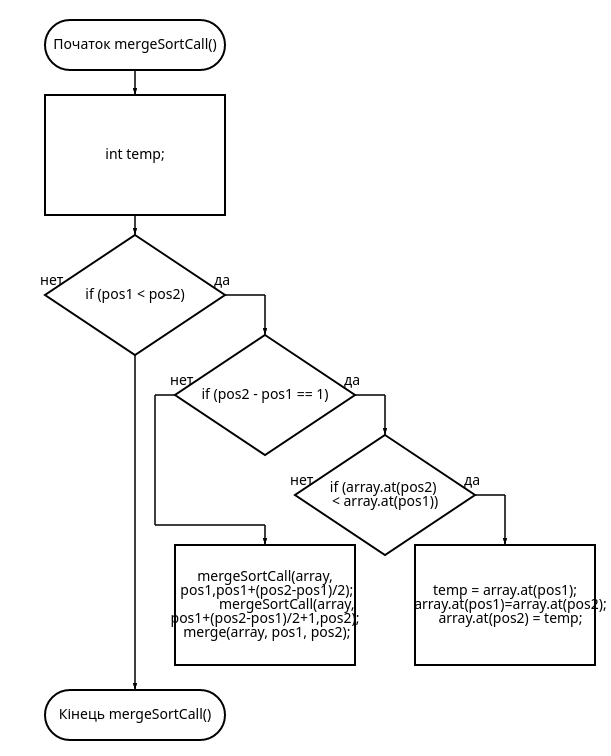
*Блок-схема 1 – fillFromFile().*



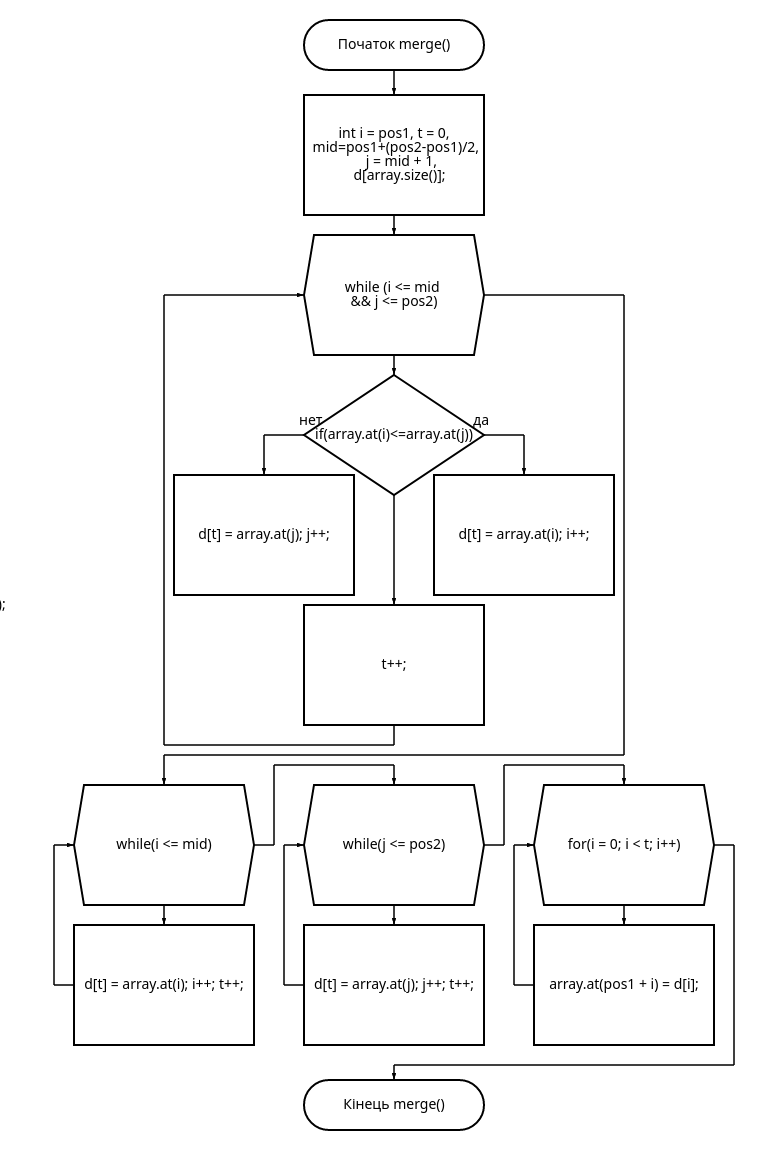
*Блок-схема 2 – saveArray().*



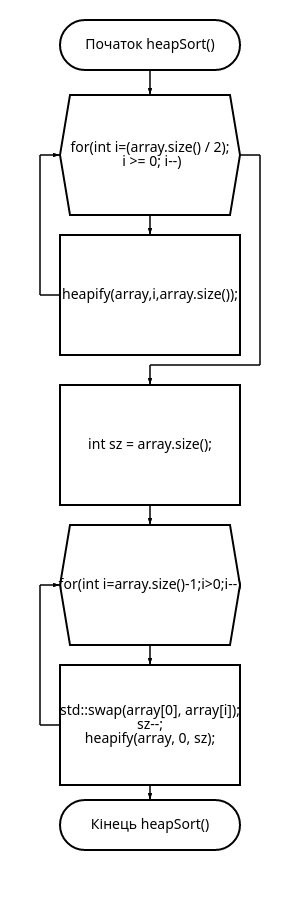
*Блок-схема 3 – quickSortCall().*



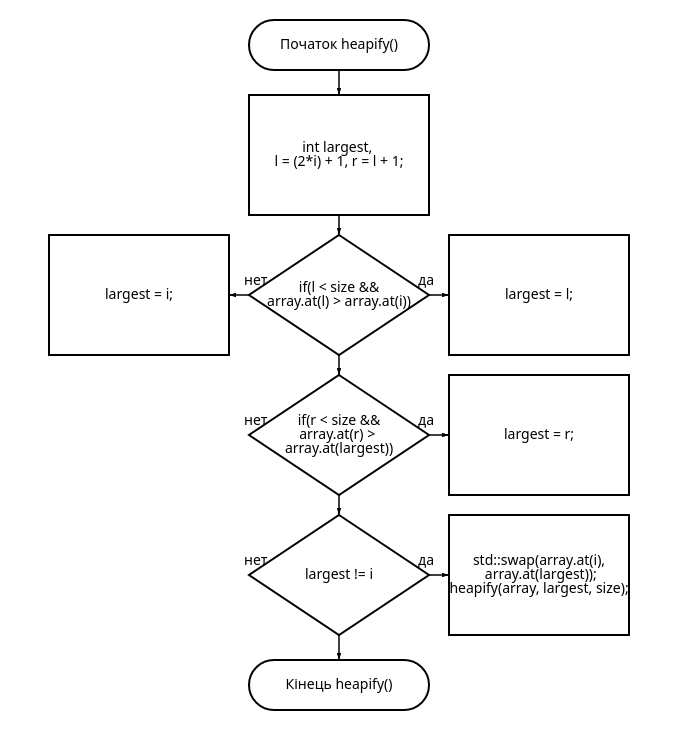
*Блок-схема 4 – mergeSortCall().*



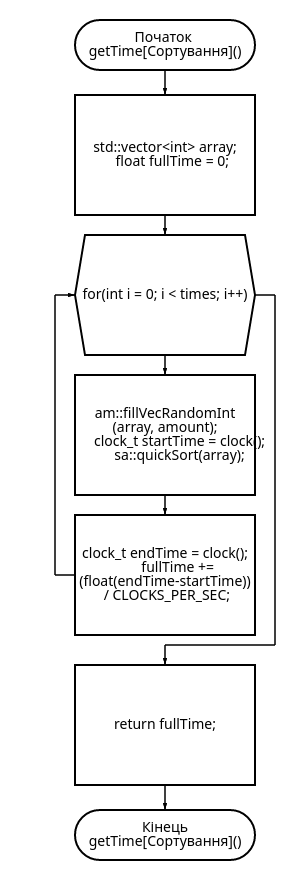
*Блок-схема 5 – merge().*



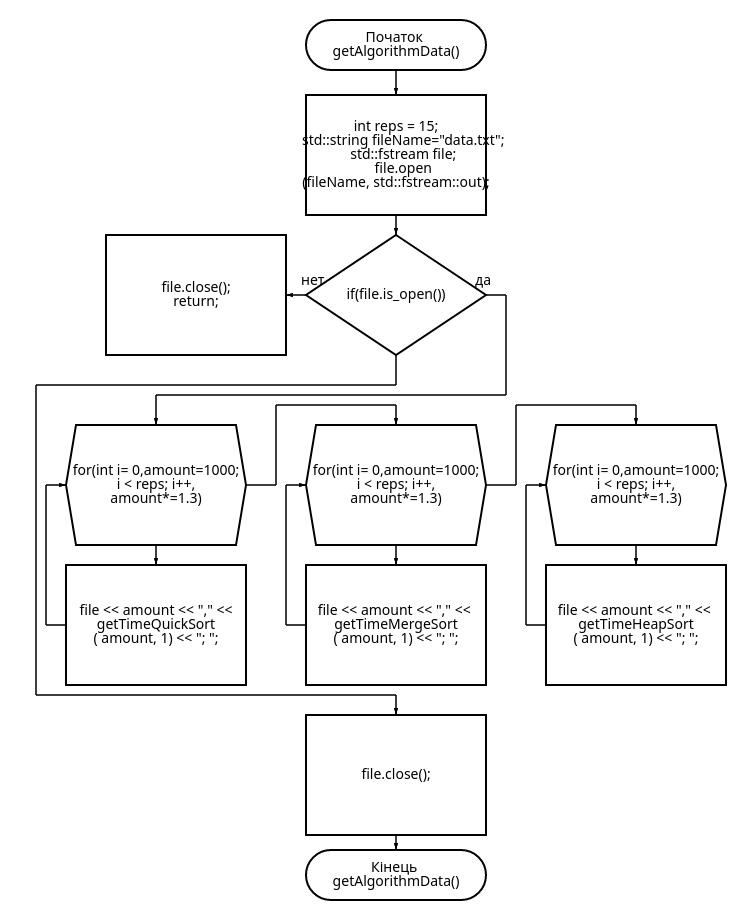
*Блок-схема 6 – heapSort().*



*Блок-схема 7 – heapify().*



*Блок-схема 8 – getTime[Сортування]().*



*Блок-схема 9 – getAlgorithmsData().*